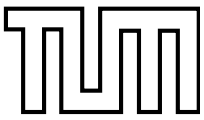

**INSTITUTE FOR REAL-TIME COMPUTER SYSTEMS**
TECHNISCHE UNIVERSITÄT MÜNCHEN
PROFESSOR G. FÄRBER


# Application Program Interface for an Embedded Linux System

Thomas Sommer

**Bachelor's Thesis**

# Application Program Interface for an Embedded Linux System

Bachelor's Thesis

Executed at the Institute for Real-Time Computer Systems
Technische Universität München
Prof. Dr.-Ing. Georg Färber

**Advisor:** Prof. Thomas Bräunl

**Author:** Thomas Sommer
Trivastr. 4
80637 München

Submitted in February 2007

Many thanks to:
Martin Hintermann for his patience and help
Prof Thomas Bräunl for creating this opportunity
Adrian Boeing for his encouragement
and the people behind all the great GNU tools for facilitating this work enormously.

# Abstract

With the advance of electronics and its ever growing miniaturization, embedded systems are more prominent than ever. They have long made their way from industrial applications into the consumer market, being contained in anything from cellular phones to automobiles. Their power has grown substantially, too. Today's embedded systems easily outperform a desktop PC twenty years ago.

However, their new complexity calls for a new design approach in turn: one that emphasizes high-level tools and hardware/software trade offs rather than low-level programming and logic design. Modularity and abstraction layers gain an important role. Development is simplified and sped up if the same concepts apply. Additionally, code can be used for multiple products, lowering the development costs significantly.

The goal of this project was the partial implementation of an interface. It will be used in programming applications for a specific embedded system: The RoBIOS library of the EyeBot M6 robotics platform. The main purpose of such an API is to make the programs written for it independent of the layers below. This way they can remain essentially unchanged even if the hardware or the operating system is modified.

The RoBIOS API and its implementation exist already and are successfully applied, however only for the predecessors of the EyeBot M6. These systems only share their purpose, the hardware and software are fundamentally different. Therefore the implementation had to be created from scratch while the interface was essentially unchanged.

Emphasis lay on routines for the user input section and the sound section of the RoBIOS library. Display routines were developed in accompanying project by a fellow student. These basic library segments allowed the creation of an application that serves as the main user interface for the robot and illustrates the functionality of the API, also being part of this project.

# Contents

# List of Figures

# List of Symbols

| | |
|---|---|
| ALSA | Advanced Linux Sound Architecture |
| API | Application Programming Interface |
| ARM | Acorn RISC Machine |
| ASIC | Application Specific Integrated Circuit |
| DMA | Direct Memory Access |
| FIFO | First In First Out |
| FPGA | Field Programmable Gate Array |
| HTTP | Hyper Text Transfer Protocol |
| GNU | GNU is not Unix |
| GPIO | General Purpose Input/Output |
| I/O | Input/Output |
| IP | Internet Protocol |
| IRQ | Interrupt Request |
| LCD | Liquid Crystal Display |
| LED | Light-Emmitting Diode |
| MB | Mega Byte |
| MMC | Multi Media Card |
| NFS | Networking File System |
| OSS | Open Sound System |
| PC | Personal Computer |
| RISC | Reduced Instruction Set Computer |
| RoBIOS | Robot Basic Input/Output System |
| SD | Secure Digital |
| USB | Universal Serial Bus |

# Chapter 1

# Overview

With the advance of electronics and its ever growing miniaturization, embedded systems are more prominent than ever. They have long made their way from industrial applications into the consumer market, being contained in anything from cellular phones to automobiles. Their power has grown substantially, too. Today's embedded systems easily outperform a desktop PC twenty years ago.

However, their new complexity calls for a new design approach in turn: one that emphasizes high-level tools and hardware/software trade offs rather than low-level programming and logic design. Modularity and abstraction layers gain an important role. Development is simplified and sped up if the same concepts apply. Additionally, code can be used for multiple products, lowering the development costs significantly.[18]

The goal of this project was the partial implementation of an interface. It will be used in programming applications for a specific embedded system: The RoBIOS library of the EyeBot M6 robotics platform. The main purpose of such an API is to make the programs written for it independent of the layers below. This way they can remain essentially unchanged even if the hardware or the operating system is modified.

The RoBIOS API and its implementation exist already and are successfully applied, how-

Figure 1.1: Embedded System Appliance: Sony Camcorder [18]

ever only for the predecessors of the EyeBot M6. These systems only share their purpose, the hardware and software are fundamentally different. Therefore the implementation had to be created from scratch while the interface was essentially unchanged.

Emphasis lay on routines for the user input section and the sound section of the RoBIOS library. Display routines were developed in accompanying project by a fellow student. These basic library segments allowed the creation of an application that serves as the main user interface for the robot and illustrates the functionality of the API, also being part of this project.

In the course of this document, the development of the API's implementation is described, from its background to its application. The first section describes the system the API is written for, the hardware as well as the operating system. Its special properties are pointed out and it finally compared to a similar system to emphasize these characteristics. The following section gives gives some background information on the concepts employed to implement the API. They refer to the hardware and software that was used but are general enough to be applicable to the development of any API for a similar system. In the last section specific details about the implementation are discussed. The employed tools are described and the workings of some parts of the library. To illustrate its use, an application using this API is described, too. Finally, some details of the Monitor program are looked at, the main user interface of the EyeBot M6.

# Chapter 2

# Hardware and Software of the Target System

The system the API was written for is a small robotics platform. It will be referred to as the EyeBot M6. Before looking at the details of its implementation, a more general look is taken at those systems.

Though not as prominent as PCs, embedded systems actually make up more than 99% of all CPUs used. They can be found almost anywhere: typical applications include engine contols, electronic testing equipment and medical instruments but also home appliances like TVs, microwave ovens or cellular phones. A typical western houshold hosts around 40 embedded systems and a modern car around 80.[3]

An embedded system can be defined as an information processing system which is embedded into a bigger technical context or technical system. More specifically, if that technical system is a device, it will completely encapsulate the embedded system and draw all its computing power from it, as opposed to a device that is connected to and controlled by an independent computer.[9]

Due to this close bond, the embedded system is typically a special purpose computing system, meaning that it was designed to perform a specific, pre-defined task and has specific requirements. This is very contrary to a general purpose computing system like PCs or Servers. The restriction of its capabilities brings along the potential to optimize it for the appointed task. In these optimizations, the sheer processing performance is usually not the main goal. Aspects like size, reliability, power consumption and cost play a more important role.

Taking a closer look at the reliability, this treat reveals the fundamentally different nature of an embedded system quite clearly. Being just a small component of the device it serves, it is not intended to be repaired or upgraded, even replacement might be difficult. This is tolerable if it is reliable enough. Continuous uptimes of years may be required and are well feasible. Physically it can be more robust than a general purpose processor due to the lower complexity of the task. This simplicity also helps to avoid logical or design

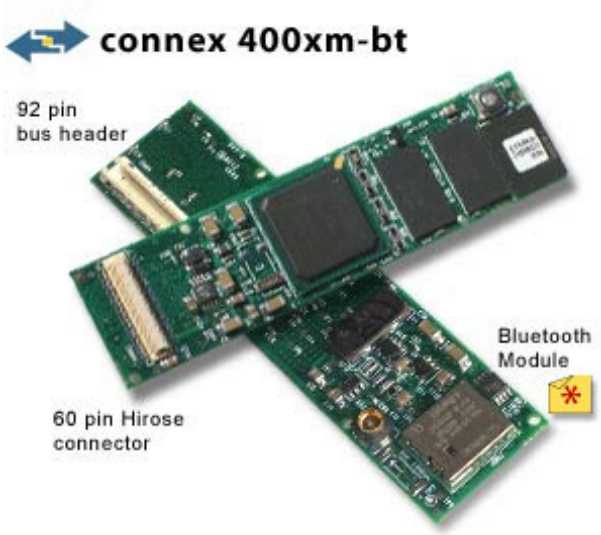


Figure 2.1: Gumstix Connex [17]

errors, but they are also minimized through a closer review. Especially with real time systems, this proof of reliability is a lot easier if not even possible in the first place.

Embedded Systems contain a software component. This makes them cheaper and more flexible than special purpose chips (ASICs). Storage can be realized on the same chip, so there would not be any difference on the outside. In fact, disc drives are avoided as they are too complex, too big and not robust enough compared to read only memory or flash memory on a chip. Though not as a whole, components of this software can be reused, making development faster and easier. Often more general software is used in the early development stages and is freed of unnecessary parts once the system goes into mass production.

Typically embedded systems do connect to sensors and actors. Human interface devices are not existent or reduced to a minimum, like LED lights and a few buttons. However with the advance of electronics more complex systems appeared, blurring the line between embedded and general purpose systems. Examples of these devices are PDAs, smartphones and navigation systems. They feature displays, keypads or touchscreens and their software can be expanded. The tight link to the hardware they control still makes them embedded systems, however.

## 2.1 The Gumstix Platform

The gumstix is a complete single-board computer that is literally the size of a stick of gum. It hosts a Linux operating system and sells for less than $200, making it popular with the open-source community and robotic system developers alike. [7]

The package measures 80 x 20 x 6.3 mm, making it well suitable for mobile applications.

A 200- or 400-MHz Intel XScale processor enable to fulfill rather complex tasks while providing a well known architecure (ARM9). Finding approproate compilers should therefore not pose a problem. Windows CE and Linux support it.

The Gumstix includes 64 MB of RAM and 4 or 16 MB of flash memory. This may seem little, but bearing in mind that embedded systems pursue very specialized tasks and that the operating system can be stripped down to the really neccessary parts, it is sufficient. If more is needed it can be added via MMC flash memory cards, for which a socket is optianlly provided. Unfortunately, SD cards, which expose the same form factor, are not supported because the open-source drivers would reveal the proprietary encryption scheme.

Usually the Gumstix single-board computer attaches to other boards using a Hirose connector, allowing access to the power source, serial ports and USB port. This way a variety of Gumstix expansion cards or application-specific custom baseboards can be attached. Single-board-computer configurations are available that replace the MMC slot with a second I/O connector to enhance expansion features. [17]

The company's website provides a a variety of software. All Gumstix configurations run the 2.6 Linux kernel and a user-space tool kit based on the embedded C library and the Busybox utilities. The software offers a complete Linux operating-system environment plus a range of open-source applications, including Bluetooth utilities, HTTP-server and -client routines, audio players, and uClibc, a small-footprint C library for embedded Linux systems. It is also possible to add a wide range of system and development tools to the Gumstix using the supplied build-root package.[16]

A possible drawback of the design is the lack of a floating point math unit. Though software emulation is possible, it will slow down processing so much that custom solutions are necessary. The ability to access the gumstix main bus directly certainly helps in this respect.

## 2.2 Operating Systems

Operating systems for embedded systems exist in many more varieties than those for PCs, for example. The main reason for this is the greater variety of the underlying hardware combined with harsher resource constraints: The operating system is custom fit to the hardware and not vice versa, as the developments in the PC sector suggest.

With embedded systems becoming ever more complex and powerful, blurring the line between them an general purpose computers, it became more attractive to make use of the established solutions for the latter. Even if modifications are needed, there remain enough advantages to building from scratch. In the following sections, two operating systems for embedded systems are outlined.

### 2.2.1 Windows CE

Microsoft's Windows CE is an operating system specifically designed for minimalistic computers and embedded systems. It is not a trimmed down version of a desktop Windows as is the case with Windows XP embedded. Rather, it features a distinctly different kernel. As a consequence, binaries from the desktop variants won't execute under Windows CE.[8]

The system is widely used in PDAs and cellular phones but also in industrial devices and embedded systems. Many technologies from the desktop variants are available making the transition easier for programmers. At the same time, it can be configured to require very little space by choosing just the really necessary among the approximately 500 components. The minimum build size is 200 KB.

The latest version (6.0) is available for a variety of architectures. It features a real time kernel, supports up to 32.000 simultaneous processes and offers an addressable memory range of 2 GB.

Unlike its desktop counterparts, it is distributed as shared source meaning that customers are able to have insight into a large part of the source code, including the entire kernel, device drivers and the file system. However, other parts remain closed. According to Microsoft, the source availability will help in providing documentation, let users add kernel space features and allow developers to share their modifications (under the shared source license).[13]

### 2.2.2 Linux

The Linux operating system, often referred to as the Linux kernel, was open source from the very beginning. In fact, this trait was the main reason for its creation and Linux became almost synonym for open source systems.

The access to the source code lead to a multitude of different flavors of the kernel, which makes it hard to characterize it as a whole. Initially written for the Intel 80386 processor, it has been ported to all major architectures. Although the kernel is monolithic, device drivers are easily configured as modules. This allows to load and unload them while running the system, saving resources and making it easier to isolate faulty drivers. This way, kernel parameters can be changed easily without rebooting the system.

The support for not-so-common hardware has traditionally been somewhat problematic. Hardware vendors were hesitant to release their code or even to provide detailed specifications of their products. With a broader acceptance of Linux this has however changed for the better and cannot be considered a key disadvantage anymore. [6]

Not surprisingly there exist approaches to add real time capabilities to the kernel, which it does not have by default. In RT-Linux, for example, a simple real time executable runs a non-real-time kernel as its lowest priority task, but other approaches have been undertaken, too. [15, 19]

Linux is a preferable choice for robotic platforms and embedded systems. Besides the features and tools it already provides, it is backed up by a large and capable developer community, offering documentation, patches and extensions. For example Gumstix Inc. attributed part of the success of their products to the help of outsiders in developing drivers. [1]

## 2.3 EyeBot M6

The EyeBot is a robotic platform that lays emphasis on image capturing and processing, hence the name. A major goal of the design was the capability to perform the processing on-board. It is still a general robotics platform, as it does not feature any dedicated logic to fulfill these tasks. Rather it offers the capabilities to handle complex tasks in real time.

Early models of this family where built around a Motorola 68332 controller. Although they fulfill the requirements and where quite successful, more complex image processing at desirably higher resolution, color depth and frame rate exposed the limits of the system.

Thus the latest model, the EyeBot M6, was a complete redesign based on the a gumstix board. The Connex 400m-bt features an 400MHz Intel XScale PXA255, 64MB of RAM, 16 MB flash memory as well as Bluetooth, USB and LCD controllers. Though this meant a considerable gain in performance, the true breakthrough lies in the incorporation of an FPGA, a Xilinx Spartan3-500E. Especially repetitive tasks that occur in image processing can be delegated to a specialized circuitry, resulting not only in a better performance but giving the CPU more time to do more completely tasks, like steering the robot.[4]

To further emphasize its purpose, the EyeBot M6 is equipped with dual cameras. This way, stereo vision is possible. A color LCD with a resolution of at least 240 x 160 pixels and a touchscreen are also noticeable improvements.

Another strength of the new design are the numerous communication ports it offers. USB and Bluetooth where mentioned before, but the EyeBot also offers serial ports, infrared and Ethernet. Audio connectors are supplied as well, and, of course connections to servo and motor controllers, encoders, position sensing devices and GPIO lines for the robotic tasks. As a mobile platform, the EyeBot may be run with batteries although the power consumption is considerable if all components are powered. [2]

## 2.4 Essex Robot

A fairly similar system compared to the EyeBotM6 is realized at the University of Essex. The goal of this project is to explore visual guidance of a small, autonomous robot. A visual sensor was chosen as sonars were considered too slow, lasers too bulky and infrared sensors too limited in their coverage. The robot pursues two tasks: reach a predefined target and avoid obstacles in the way. While the first task can have a low priority, the

second must be performed in real time, since the safety of the robot is directly affected. The obstacle detection is performed omnidirectional by means of a spherical mirror and a camera. Processing this data is fairly complex, therefore the developers chose a Gumstix system in favor of a microcontroller. Larger and more power-hungry systems were not an option since the mobility of the system needed to be maintained.

All image processing is done by the CPU. To transfer the data fast enough from the camera, it's FIFO is directly attached to the CPU bus, which the Gumstix provides for in some of the Connex models. Some more quirks were necessary to ensure real time processing: For once, the kernel memory mapping is bypassed through the use of a custom built driver. The data transfer is handled by the hardware DMA modules alone, not disturbing the CPU. Furthermore, a fast IRQ architecture called FIQ, which is supported by the ARM processor of the Gumstix.

With these measures, the system was successful in processing the images of the 320x240 resolution camera in real time, prerequisite to avoid the obstacles. The remaining processing capacity could be used for steering and other tasks. The team stated, that the use of the Gumstix greatly sped up the development due to the USB connectivity and the NFS support of the board. This provides for a simple and fast transfer of files, which is important in the development phase.[7]

# Chapter 3

# Employed Concepts

## 3.1 System Calls

System calls are requests to the kernel to provide a certain service. This class of routines cannot be called directly as uncontrolled access to them by user space applications would greatly compromise the stability of the system. For example, sending data to the hard disc drive by writing to a certain region in memory is not only too low level to be practical but could also damage hard and software.

Instead, the hard disk driver or the kernel itself provide a routine that receives bytes and writes them to disk. These routines would be specific for each driver, as naturally very different actions have to take place before these bytes are stored on the device. Also, only the kernel would be allowed to execute them. In order to give user space programs access to the device, the kernel provides routines that operate on the device. They are called system calls to distinguish them from ordinary library calls. For example the write system call will select the appropriate write routine depending on the device and issue the write.

Well over 300 different system calls are defined on a recent Linux, but only a few are needed when communicating with devices. Obviously reading and writing is needed, covered by the system calls read and write respectively. The data involved in these calls will always be outputted, so to control the device, the system call ioctl is used. It covers all routines that are to device specific to have an own system call. Finally open and close should be mentioned, that grant or terminate access to a device.

One may ask why system calls are necessary here when they are rarely seen elsewhere. The answer is simple: library routines use system calls internally, so they are not apparent to the programmer. When creating a library, however, one cannot just resort to other library routines. The performance would be low and the need for the library would be questionable when it is only redefining already existing routines. When dealing with custom built device drivers, libraries usually don't exist for them anyways.

Apart from using system calls to communicate with devices, they can be utilized to get information about the state of the kernel, too. Processes, for example, are managed by the kernel. Querying the kernel about them would naturally be the most direct and thus fastest way.

## 3.2 The Proc Filesystem

### 3.2.1 Definition

The /proc filesystem is a facility that permits communication between kernel and user space, thus allowing user space programs to access kernel data as well as to change the runtime state of the kernel. In accordance to the Linux philosophy that puts files as a cornerstone of organisation, the /proc filesystem is realized as a hierarchical set of files, as the name suggests. However, it is a pseudo-filesystem, meaning that the files are not located anywhere on disk or in memory but exist only a reflection of kernel data structures. This is possible by by exploiting the virtual filesystem layer Linux provides.

### 3.2.2 Application

This approach makes it quite easy to access the desired data: Ordinary tools that read from files can be applied to extract information and consequently tools that write to files are enough to manipulate the current settings. The following example shows how to use the cat command to read out the current setting of whether the system is forwarding IP datagrams between network interfaces:

```
\# cat /proc/sys/net/ipv4/ip_forwarding
```

In this case, the output consists of a single digit: 1 if the property is set and 0 if it is not set. This makes writing to the file rather easy - the echo command will be sufficient to set the entry to say 1 to enable it:

```
\# echo 1 > /proc/sys/net/ipv4/ip_forwarding
```

Unlike settings in config files, this change will take effect immediately as it is written directly to the corresponding memory position. As said before, the files of the proc filesystem are not a copy of the current state nicely refined to make access easier. They don't have to be scanned in again for the kernel to acknowledge the changes. Rather they represent an interface. This is expressed, too, by the fact that all these files have a size of 0 and a modification time that is always the current time.

### 3.2.3 Pros and Cons

Thus the /proc filesystem offers the advantages of having fast and easy access to kernel data. Fast, because the fact that files are used is just representation - essentially the kernel is communicated with directly. Easy, because the data is organized hierarchically and independently of the names and forms of the underlying data structures. This is an important point for the portability of programs that use this data.[12]

However, making use of the /proc filesystem goes along with some disadvantages as well. The most striking would be that the facility is relatively new and does not impose strict standards or naming conventions. While this may be acceptable for a real person trying to get some information, it poses problems for programs that rely on it and should be portable.

But even if the data can be found reliably, it must be parsed. Many files contain not just a single value but a whole table. Extracting values from such files is certainly more error prone and time consuming than receiving a structure as when using appropriate system calls.

At last, security issues shall not be overlooked: Manipulating the filesystem (not the /proc filesystem itself but the root filesystem, mount points and standard paths) could be used to intercept or change the data read via /proc.

## 3.3 Input Interface

The input interface under Linux is covered here, as it serves as an example that accessing the devices directly is not always the preferred way.

User input usually does not come from a single source. A standard desktop PC offers at least a keyboard and a pointing device like a mouse or trackball. And even if the mouse is not used, modern keyboards are often recognized as multiple distinct devices when they feature additional keys. Those keys usually provide faster access to often used functions like copy and paste or email and cannot be handled by the standard driver. Even the buttons on monitors can form an input device.

It would be a nuisance if an application would have to know about all these different drivers. Instead, it expects input events and is not actually concerned what physical device generated them. It does so by communicating to event handlers, which in turn communicate with a single entity, the input core driver. It unites feeds from all specific input device drivers and is the only one that communicates with them directly.[11]

User space programs don't need to access the input core driver if they make use of the generic event interface. This facility will create a device node for each input device that uses the input core. They are named event0 ... event31 and reside under /dev/input/. Moreover, the event driver registers itself as an event handler. As a result, user space programs can read(2) on the file descriptor and will receive the input event generated by

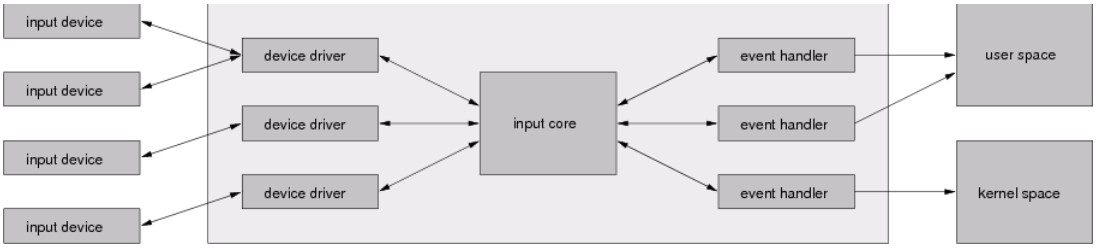


Figure 3.1: Linux Input System [11]

the corresponding device in the form of input_event structures. Their layout is specified in section 4.

This indirect access of the input device drivers offers the advantages mentioned above without requiring a big and time consuming framework.

## 3.4 RoBIOS M6 Library

The RoBIOS Library for the EyeBot M5 contained the following sections and so will the version vor the M6.

- Image Processing
- Key Input ⋆
- LCD Output
- Camera
- System Functions ∘
- Multitasking
- Semaphores
- Timer
- Download and RS-232
- Audio ⋆
- PSD Sensors
- Servos and Motors
- V-Omega Driving Interface
- Bumper / Infrared Sensors
- Latches
- Parallel Port

- A/D Converter
- Radio Communication
- Compass
- TV Remote Control

(∘: partially implemented)

A detailed list can be found in the EyeBot Online Documentation. [14]

It shall be mentioned that the implementation of the complete RoBIOS library would suffice for several such projects. Here, only a portion of it was to be implemented (⋆). Another part was assigned to fellow student Martin Hintermann, specifically the LCD routines. He also contributed substantial parts to the Monitor program and his work in setting up the cross compiler and the EyeBot made this project feasible in the fist place.[10]

# Chapter 4

# Implementation

## 4.1 Development Tools

Gumstix provides not just a compiler for its boards but also an entire toolchain. These tools are GNU and run best under a Linux or at least a POSIX operating system. Although they are ported to Windows, too, the absence of a powerful shell and the different file system greatly diminish their usefulness.

Thus the entire development was done under a Linux system. This offered another major advantage: Since the target system runs Linux, too, the code could be compiled for a standard PC and executed nicely. Of course, the hardware differences forbade reliable tests, but for developing purposes, this method proved useful.

In order to compile the code on any platform, the make tool was used and appropriately configured. With the created setup, code specific configuration files could be reduced to a minimum. All settings are derived from the path of the code and the naming scheme of the files. This way, cross compiling is reduced to copying the files to a certain directory or by changing the directory's name.

## 4.2 Key Routines

As stated before, the user input device of the EyeBot M6 differs greatly from its predecessor. Yet it was still desired to change the RoBIOS routines only as much as necessary, and, if possible, emulate the behavior of the EyeBot M5. The following section shows how this was achieved.


Figure 4.1: EyeBot M5: menu and buttons: [14]

### 4.2.1 Keycodes

The EyeBot M5 featured four buttons located at the lower edge of the display. This way, their current function could be made visible to the user by displaying a descriptive label in the last line of the display, the so called menu. As the number of keys was fixed, so was the size of the menu and, more importantly, the representation of the keys in the code.

The routines of the old RoBIOS implementation now returned an integer value representing the key. Each key corresponded to a bit in this value: if it was set, the key had been pressed. So these functions were able to communicate the state of all keys in a single value. However this feature was rarely exploited.

The EyeBot M6 does not have physical keys, it relies on the touchscreen alone. The use of a touchscreen certainly has the advantage of being much more flexible and intuitive: Arbitrary numbers of keys can be defined and visualized on the display. The user can directly press these virtual keys, not a key in their vicinity. Moreover input methods that do not relate to keys at all can be implemented, for example scroll bars.

Along with this flexibility comes a greater complexity. With the resolution of the touchscreen being similar to that of the display, hundreds of keys could be specified. This does not just include keys that relate to a single pixel: Usually a range of pixels should be linked to a single key, so a key is represented by a list of these pixels or, more generally, by a geometric description of them. For example, the first menu button could be specified by a rectangle int the lower 10 percent of the screen extending from the left border to 25 percent of the width. While this is feasible, the type of the keycode variable would have to be replaced by something that can hold this information - an integer would not be enough.

Instead of modifying the API this drastically, an intermediate variable was introduced. It helps to resolve the location and shape of a key into an identifier, for simplicity a positive

integer. Depending on this value, a bit is set in the keycode. With an integer range if 32 bit, the state of roughly 30 keys can be represented in the keycode of that type (some bits may want to be reserved to indicate an invalid keycode and such). Recalling that a even a standard PC keyboard has only about 100 keys, this limit seems reasonable, especially since the dimensions of the touchscreens are such that only about 20 keys can be accessed by a finger tip. If still more keys are needed, extending the keycode variable type to say 64 bit is an easy adjustment to the code.

### 4.2.2 Touchmap

As mentioned above, the use of a touchscreen required a mechanism to translate the data received into key identifiers. The touchscreen driver will generate input events that contain coordinates. More specifically, these data if these events can be interpreted as the following structure, defined in linux/input.h:

```
struct input_event {
     struct timeval time:
     __u16 type;
     __u16 code;
     __u16 value;
     };
```

Of main interest are the events that contain absolute coordinates of the area that is touched. These values are relative to the touchscreen's coordinate system and generated from analog signals. As it makes no sense to calculate with a a higher resolution that that of the screen, they are converted to screen coordinates, fist. This includes a rotation of the screen by 180 degrees, as that the display of the EyeBot M6 is used upside down. Also, their analog origin requires that the values are capped.

Certainly the most flexible way to specify which coordinates belong to which key would be to maintain a list of geometrical objects. They would be relative to the screens dimension to be independent of the hardware. To test which key was pressed, a check would be performed whether the point lies in the boundaries of such an object.

However, this approach adds a lot of complexity to such a minor task. Even for rectangular key regions, four comparisons are necessary. Triangles would require some calculations and circles would even call for floating point math, which is to be avoided whenever possible. To avoid gaps or overlaps, a lot of care would have to be taken when creating these objects.

Instead, a lookup table was opted for, referred to as the touchmap. It can be viewed as a two-dimensional array that represents the screen. Each entry is equal to the key identifier of the key at that location. Realized as an array of bytes, 256 different identifiers can be addressed, far more than the keycode can hold. The dimensions of the table are variable, which allows to adjust them to just the size that is needed.

The menu of the EyeBot M5 is located in the last text mode line. Turning it into touchable keys, one could specify the text mode resolution as the dimensions of the touchmap. Thus every character cell would correspond to a tochmap cell. Filling the first cells of the last line with the value 1 will enable the fist menu key. In review, it becomes clear that horizontally only a resolution of 4 is required. Even if now the border of the touchcells might not be aligned with that of the character cells, that just shows the limitations of the text mode: after all, the menu items should be evenly distributed. Going a step further, one can reduce the vertical resolution to say 2, allowing any touch in the lower half of the screen to hit the menu. This may seem inaccurate, but again, only the fact that the menu shared the space with the rest of the displayed information made it so small. The touchscreen is transparent so this limitation does not apply.

Specifying a touchmap larger than the text mode resolution may seem unnecessary. After all, these cells would be hard to pick with a stylus, not to mention a finger tip. Still there is use for it: A touchmap could be created automatically from a picture, only the relation of what color corresponds to what key would be needed. Thus it is easy to implement arbitrarily shaped key regions or touchmaps created in during runtime.

A reverse effect is, that the location of the keys can be displayed easily and fast. This way, a menu can be done without when the key location is intuitive enough. The keys would be invisible but could be displayed when help is required. A steering cross for panning a picture would be an application for this.

### 4.2.3 List of Routines

| NAME | DESCRIPTION |
|---|---|
| KEYGetBuf | wait for keypress and write keycode into buffer |
| KEYGet | wait for keypress and return keycode |
| KEYRead | read keycode an return it (nonblocking) |
| KEYWait | wait for a specific keypress and return it |
| KEYGetXY | wait for keypress an return screen coordinates |
| KEYSetTM | configure the touchmap |
| KEYGetTM | get a string representation of the touchmap |

### 4.2.4 Example

The following code illustrates the use of the keycode. A message is printed depending on the pressed key.

```
      keycode_t keycode;
      keycode = KEYGet()
      if(keycode == 8);
        LCDPutString("key 3 pressed");
i
```

or more intuitive:

```
#define KEY3 1<<3
keycode_t key;
key = KEYGet()
if(key == KEY3)
  LCDPutString("key 3 pressed");
```

more advanced:

```
keycode_t keycode;
keycode = KEYWait(1<<3 | 1<<4)
if(keycode == 1<<3)
  LCDPutString("key 3 pressed");
else
  LCDPutString("key 4 pressed");
```

## 4.3 Audio routines

The general principle of the audio routines is simple: The right device file is opened either in read or write mode. Then ioctl system calls are used to configure it and finally the data is written to or read from the device. However, some additional aspects had to be taken care of in the implementation.

### 4.3.1 Device capabilities

First of all, the capabilities for the device have to honored. This may sound trivial, but it has to be kept in mind, that by communicating with the device file, one does not access the hardware directly. A setting must be supported by the hardware module and the sound API. Usually a request for a certain setting returns the value to which the device was set to. The standard values are usually accepted and if invalid values are passed, the device will pick the closest valid one. In a general sound application, no assumptions on the sound cards capabilities must be made. Thus every setting should be checked and appropriate measures should be provided if it fails. But as the library targets a specific system these steps were left out, to simplify and speed up the code. The current implementation supports only a few standard settings, as an thorough survey of its capabilities was not undertaken.

### 4.3.2 File format

The old RoBIOS was able to handle various file formats. This capability was dropped or at least postponed in order to have a simpler interface and sleeker code. After all, it is

not vital for the EyeBot to master multitudes of sound formats, its main application lies in image processing. Due to the limited space, sound files should always be converted to the lowest acceptable quality when transferring them to the EyeBot.

The WAV file format was chosen as the format for sound samples and files alike. It is commonly used and platform independent. The standard is very flexible, in fact, the WAV format are a subset of RIFF format that allows much more than just sound contents. [5]

In order to speed up the processing of the files, it was made a requirement that they confirm to a specific layout. It is referred to as canonical RIFF-WAV-PCM in the code and is well suitable for simple sound files. Most ordinary WAV files comply to it, although the WAV format does not demand this.

### 4.3.3 Nonblocking Playback and Exclusive Access

It was desired to have any lengthy RoBIOS routine be non-blocking, i.e. it will execute its work in the background and allow the calling program to execute the next command. Audio playback and recording certainly fall into this category. Special routines exist to check on their status, to avoid simultaneous access to the sound device or simply to start processing the result.

This functionality was implemented with threads. Global variables exist to represent the state of a sound playback, a sound recording or a tone playback. They are set once the corresponding routine successfully gets access of the device and reset by the thread. As usual, care must be taken to avoid concurrent access to these variables. The pthread library that comes with the Linux kernel offers the necessary tools.

### 4.3.4 List of Routines

| Name | Description |
|---|---|
| AUPlaySample | Plays an audio sample. |
| AURecordSample | Records an audio sample. |
| AUPlayFile | Plays an audio file. |
| AURecordFile | Records an audio file. |
| AUCheckSample | Checks for audio playback end. |
| AUCheckRecord | Checks for audio recording end. |
| AUCheckTone | Checks for tone end. |
| AUTone | Plays a tone. |
| AUBeep | Plays a short tone. |
| AUCaptureMic | Probes the microphone input. |

## 4.4 Example Application

This small example illustrates the use of the RoBIOS library. It is essentially a beefed up version of the classic "'Hello World"' that will output a message on the screen. Here a tune is played too and the program waits for the user to touch the screen. In fact, this is necessary to hear the sound at all: The play thread would terminate if the main program exited too fast.

```
#include "eyebot.h"

int main (void)
{
  LCDInit();
  KEYSetTM(M6KEY_CLASSIC);
  AUPlayFile("nicetune.wav");
  LCDSetString ("Nice tune playing ...");
  KEYGet ();
  KEYSetTM(M6KEY_RESET);
  LCDRelease();
  return 0;
}
```

For compilation, the library needs to be included via -leyebot. Also, the wave file is expected in the working directory...

## 4.5 Monitor Program

### 4.5.1 Purpose and Structure

The purpose of the so called Monitor program are twofold. For once, it should demonstrate the capabilities of the platform and the use of the API. Secondly, it should provide a base for controlling the EyeBot and launching other programs.

The program is started automatically after the bootup completes. As the user has no means of text input, no login must be required and all other available programs should be located at predefined places, so they are accessible easily. Normally, exiting the program should not be provided for, though it may prove useful to allow termination just to respawn it immediately and thereby essentially resetting it.

The current implementation is heavily modular. The building blocks are referred to as pages, as they usually clear the screen and display a new content. Pages cannot communicate with each other and only share a single global variable. This is a table of the currently available keys, together with a description of them and a pointer to a page.

It allows displaying help on the available keys and a more elegant handling of the key presses.

Other global variables are implicitly set, like properties if the display or the touchmap. This is important since pages may change them. When no standard values are defined, a page will have to save the current state and reload it when it finishes.

A repeating element in the Main Program is a list of items of which the user selects one. As the list generally won't fit on the display, it is scrollable in a sense, that it flips to the next section. An indicator marks the current entry. that indicator can be moved via menu buttons or directly via the touchscreen.

This facility is realized as a separate routine, which not only saves resources but also introduces a common look and feel. For the developer, it allows fast and easy creation of such lists. It should be noted that the KEYSetTM routine has a mode just for this application, so this list illustrates its use.

### 4.5.2 Page Example

In this example, a help page is created. It clears the screen, outputs some text and waits for user input. The page has its own menu, so the old keys are saved and restored afterwards. In this special case, the second menu button is empty. Therefore, only certain keys are accepted when the user input is queried.

```
int page_help (void)
{
  keylist_t *orig_keylist_p = Keylist;
  keycode_t keycode, valid_keycode;
  int key;

  Keylist = create_keylist (4);
  set_key (1, "BACK", "previous page", run_nothing_func);
  set_key (3, "MAP", "key region map", page_touchmap);
  set_key (4, "LIST", "key list", page_keylist);
  valid_keycode = 1<<1 | 1<<3 | 1<<4;

  do
    {
      LCDClear ();
      LCDMenu (Keylist->key[1].label, Keylist->key[2].label,
               Keylist->key[3].label, Keylist->key[4].label);
      LCDSetString (0, 0, "Help-Page");
      LCDSetString (2, 0; "Bla Bla ...");
      keycode = KEYWait (valid_keycode);
      for (key = 0; keycode >> (key + 1); key++);
```

```
      Keylist->key[key].run_func ();
    }
  while (key != 1<<1);

  free (Keylist);
  Keylist = orig_keylist_p;
  return 0;
}
```

### 4.5.3 Process Handling

The Main Program can execute arbitrary other programs. It reads directory contents and offers these files to the user. Of course, the vast majority of programs is not suitable, as they either require arguments or they output to standard streams, which will not be displayed in the Main Program. Still the possibility to create simple shell scripts that give access to the Linux tools is welcomed, if only as a temporary solution.

The execution of the Main Program will not be blocked by these calls. Rather, the processes can be run in the background. They are added to an internal process list, which can be brought up to check the status or forcefully terminate a process.

# Chapter 5

# Conclusion

In the course of this project, substantial parts of the RoBIOS M6 library were implemented and an application for it was created. In preparation of this work, the EyeBot was set up and development tools were configured. This will facilitate future work on the system.

The implemented parts of the library include the key input section, the audio section and some routines for the OS section. The code is thoroughly documented, including shortcomings. For every section, a test application is provided. It allows the developer to verify that the functions work as they should. Together with the LCD section, these routines provide a foundation for the RoBIOS M6 library.

With the Monitor program, an application was created to demonstrate the potential of both hardware and library. This application also serves as the user interface for the robot. It is highly modular to allow easy adjustments and extensions once other sections of the library are implemented.

## 5.1 Shortcomings

Not all goals of this project were met. In general, the code certainly leaves room for optimizations and some techniques may have to be reconsidered.

Specifically, the use of the OSS interface for the audio routines may be inappropriate - ALSA could be the better choice. It has officially replaced OSS in the kernel. However, the advanced functionality ALSA offers is not needed for the EyeBot. This and the fact that OSS is more established and mature than ALSA lead to the current situation.

The *Tone routines will not work with the installed kernel. They require a console, which is possible with newer versions. Besides that, the AUCaptureMic routine was not implemented.

The key input routines may deserve a mode in which the event device is read out constantly. Thus it would not have to be opened upon every invocation and key states

could be determined. Whether the extra resources justify that method would have to be investigated.

## 5.2 Future Work

Future work will certainly include the completion and optimization of the library. In this course, the interface should be reevaluated to see what parts of it are not suitable anymore and what new parts will have to be introduced.

Once the library is complete, the naming of constants and globals should be revised. Currently some identifiers are rather long and comply to different naming schemes. Also, standards on return values and error messages should be imposed to display a common behavior.

Apart from that, there will hopefully be many applications for this library, making the EyeBot M6 as successfully as its predecessors.

# Appendix A

# RoBIOS Routines

## A.1 Key Routines

### keycode_t KEYGet (void)

Wait for a keypress and return a keycode. (See libM6key-types.h for a definition of keycode.) Basically a wrapper for KeyGetBuf.

Returns: The keycode.

### int8_t KEYGetBuf (keycode_t * *buf*)

Wait for a keypress and store the keycode into the buffer. (See libM6key-types.h for a definition of keycode.)

Parameters:

*buf* A pointer to the variable the keycode is written to.

Returns: 0 on success, -1 on failure

### void* KEYGetTM (int8_t *mode*)

Returns a printable representation of the current touchmap. The needed space for the return value is allocated and should be released by the calling function!
Supported modes:
M6KEY_TOUCH String with key index at nearest positon only.
M6KEY_TEXT String with key index at all positions. For multi-digit indices, a conversion is used. Currently: A..T for 10..29, then cycle.
M6KEY_POINTER a pointer to the underlying structure. (do not mess with it - use it merely to save the current state and restore it later)

Globals used: m6_touch_p (internal)
Prerequisites: LCDInit()

Returns: void* (char* or touchmap_t*)

### coord_pair_t KEYGetXY (void)

Wait for a keypress and return the correspnding coordinates. This makes sense only if a touchscreen is the source of key events.

Returns: The coordinates (x,y). (See libM6key-types.h for a definition of "coord_pair")

### keycode_t KEYRead (void)

Check whether a key is pressed and return its keycode. Unlike KEYGet etc it will return after a short time even if no key is pressed. In this case, a special "invalid" keycode is sent. This function does not read out which keys are pressed (held down) at the moment i.e. it does not query the state of the keys. It will only acknowledge actions that took place after the function was called. The query state functionality is not available, as the Eyebot M6 uses a touchscreen that does not generate "key events" but merely coordinates. It would still be possible to realize this in a function, but modifying the touchscreen module is the cleaner way... (See libM6key-types.h for a definition of keycode.)

Returns: The keycode of pressed (i.e. pushed down and held) keys is returned or 0 if no key is pressed. Warning: if you use a touchmap that contains "empty key" entries (index 0), you will not ba able to distinguish between "empty key pressed" and "no key pressed". This is ok, because thats what the key is for...

### int8_t KEYSetTM (int8_t *mode*, ...)

Sets the global "key region map" where the location and the values of each key are stored. Entries in this map are what the other KEY* functions use for return values.
Globals used: m6_touch_p
Supported modes:
M6KEY_RESET frees the memory allocated for the "key region map"
M6KEY_CLASSIC: 1..4 on lower half
M6KEY_CLASSIC2: 1..4 on lower half, 5..8 on upper half
M6KEY_3ROWMENU: 1..4 on lower half, 5..X in text mode rows More precisely, 2 title rows are spared, followed by 3 rows per key. If rows remain between last bundle and menu, they are set to 0 like the title rows.
M6KEY_POINTER: set "key region map" to the supplied pointer. This is could be used when saving and restoring the map.
Globals used: m6_touch_p
Prerequisites: LCDInit() for 3ROWMENU

Parameters:

*mode* one of the modes explained above.

... additional parameters depending on each mode.

Returns: 0 on success, -1 on failure See libM6key-types.h for a definition of "coord_pair"

#### keycode_t KEYWait (keycode_t *excode*)

Wait for a specific key configuration, i.e. until all keys that are contained in the keycode "excode" are pressed. (maybe others, too!) To wait for a certain key press and nothing else, use KEYWait and evaluate the result until it matches "excode" exactly. (See libM6key-types.h for a definition of keycode.)

Parameters:

*excode* The keycode of the keys expected to be pressed.

Returns: keycode of ALL pressed keys.

## A.2 Audio Routines

#### int AUBeep (void)

Generates a short tone (beep).

Returns: '0' always.

#### int AUCaptureMic (void)

Grab current microphone input value.

Returns: ?

#### int AUCheckRecord (void)

Checks whether there is an ongoing recording. Will only detect recording by AURecord* routines.

Returns: '0' if recording takes place, '1' if not.

### int AUCheckSample (void)

Checks whether there is an ongoing playback. Will only detect playback by AUPlay∗ routines.

Returns: '0' if playback takes place, '1' if not.

### int AUCheckTone (void)

Checks whether there is an ongoing beep or tone. Will only detect playback by AUTone and AUBeep routines.

Returns: '0' if recording takes place, '1' if not.

### int AUPlayFile (char ∗ *file*)

Plays a sound sample from a file. This is no wrapper for **AUPlaySample()**, as the data transfer is buffered.

Parameters:

*file* file name the data is read from. (canonical WAV/PCM)

Returns: '0' on successfully preparing playback - the actual writing may fail, though. '-1' on failure.

### int AUPlaySample (char ∗ *sample*, long *length*)

Plays a sound sample. The actual writing to the audio device is handled in a separate thread, so the playback is non-blocking. The function will fail if another sample is played (via the libM6au routines). The audio parameters are fixed to: 8 bit, mono, 11025 sample rate, pcm.

Parameters:

*sample* defined as the contents of a canonical WAV/PCM file (for details, see libM6au-types.h)

*length* length of the sample in bytes.

Returns: '0' on successfully preparing playback - the actual writing may fail, though. '-1' on failure.

### int AURecordFile (char * *file*, long *length*, long *sample_rate*)

Records a sound sample. This is no wrapper for **AURecordFile()**, as here the data transfer is buffered.

Parameters:

*file* file name the data is written to. (canonical WAV/PCM) implicit settings: 16 bit, 1 channel

*len* bytes to sample, including the header (44 bytes).

*sample_rate* samples per second.

Returns: '0' on successfully preparing recording - the actual recording may fail, though. '-1' on failure.

### int AURecordSample (char * *buffer*, long *length*)

Records a sound sample into the provided buffer. The audio parameters are fixed to: 8 bit, mono, 11025 sample rate, pcm. (see top of this file)

Parameters:

*buffer* memory region that receives the sample, which is defined as the contents of a canonical WAV/PCM file, (for details, see libM6au-types.h)

*length* bytes to sample, including the header (44 bytes).

Returns: '0' on successfully preparing recording - the actual recording may fail, though. '-1' on failure.

### int AUTone (int *freq*, int *msec*)

Generates a tone. Creates a thread to prevent blocking execution of the main process.

Parameters:

*freq* tone frequency in Hz.

*msec* tone duration in ms.

Returns: '0' on successfully preparing recording - the actual recording may fail, though. '-1' on failure.

## A.3 OS Routines

### info_cpu_t* OSInfoCPU (void)

Collects appropriate information and returns a filled structure.

Returns: pointer to a structure (see libM6os-types.h for details)

### info_mem_t* OSInfoMem (void)

Collects appropriate information and returns a filled structure.

Returns: pointer to a structure (see libM6os-types.h for details)

### info_misc_t* OSInfoMisc (void)

Collects appropriate information and returns a filled structure.

Returns: pointer to a structure (see libM6os-types.h for details)

### info_proc_t* OSInfoProc (void)

Collects appropriate information and returns a filled structure.

Returns: pointer to a structure (see libM6os-types.h for details)

### char* OSVersion (void)

Returns the current RoBIOS version as a string.

Returns: string containing version number (see libM6os-types.h for details)

# Bibliography


[1] Anderson, Don: *A one year report.* paper, 2005. 7

[2] Blackham, Bernard: *The development of a hardware platform for real-time image processing.* paper, 2006. 7

[3] Cherba, Mike: *Introduction to embedded linux.* EUGLUG Presentation Series, 2005. 3

[4] Chin, Lixin: *Fpga based embedded vision systems.* paper, 2006. 7

[5] Corporation, IBM and Microsoft Corporation: *Multimedia programming interface and data specifications 1.0.* paper, 1991. 20

[6] Garrels, Machtelt: *Introduction to linux.* paper, 2006. 6

[7] George Francis, Libor Spacek: *Linux robot with omniderectional vision.* paper, 2006. 4, 8

[8] Hall, Mike: *Comparing windows ce and windows xp embedded.* www.embeddedtechjournal.com, 2005. 6

[9] Herkersdorf, A.: *Hw/sw-codesign introduction.* lecture script, 2006. 3

[10] Hintermann, Martin: *Development Tools for an Embedded Linux System.* paper, 2007. 13

[11] Hönig, Timo: *Input Abstraction Layer.* paper, 2005. iii, 11, 12

[12] Jonathon T. Griffin, George S. Kola: *Linux process controll via the file system.* paper, 2005. 11

[13] LinuxDevices.com: *Microsoft opens full windows ce kernel source.* online journal, Nov. 2006. 6

[14] robotics.ee.uwa.edu.au/eyebot/: *Eyebot online documantation.* www, 2007. iii, 13, 16

[15] Straumann, T.: *Open source real time operating systems overview.* paper, 2001. 6

[16] Webb, Warren: *Tiny computer holds embedded treasue.* www.edn.com/contents/images/6363909.pdf, 2006. 5

[17] WWW.GUMSTIX.ORG: *Connex product details.* online documentation. iii, 4, 5

[18] WWW.SONY.DE: *Sony consumer products.* online documentation. iii, 1

[19] YODAIKEN, VICTOR: *The rt-linux approach to hard real-time.* paper, 1997. 6